The CSI Journal on Computer Science & Engineering

The CSI Journal on
Computer Science and Engineering
Vol. 10, No. 2 & 4 (b), 2012
Pages 10-19
Regular Paper

On-Line Control Flow Error Detection and Correction Based on Monitoring Both Data-Flow and Control-Flow Graphs

Mohammad Maghsoudloo Hamid R. Zarandi Navid Khoshavi

Department of Computer Engineering and Information Technology, Amirkabir University of Technology, Tehran, Iran

Abstract

This paper proposes two efficient online techniques, Control-flow and Data Errors Correction using Data-flow Graph Consideration and Miniaturized Check-Pointing, to correct control-flow errors. These techniques are based on: 1) addition of redundant codes in a given program; 2) monitoring both data-flow and control-flow graphs of the program to detect and correct control-flow and data errors. The major importance of these techniques is the ability to recover data errors, arisen due to occurrence of the control-flow errors, with imposing less performance and memory overheads compared to previous techniques. In order to evaluate the proposed techniques, a functional simulator is used, and behaviors of five well-known benchmarks are studied on a simple processor with a real operating system. The experimental results demonstrate that at least 93% and 90% of control-flow errors can be detected and corrected by CDCC and MCP, respectively. An optimization approach is also introduced to decrease memory and performance overheads.

Keywords: Control-Flow Error, Control-Flow Checking, Error Detection, Error Correction, Check-Pointing, Data Replication.

1. Introduction

According to Moore's law, continuous device scaling enabled by modern fabrication technologies has provided smaller transistors with lower threshold voltages and tighter noise margins [1], [2]. Driven by these advances in CMOS semiconductor technology, the steady microprocessors performance has been recently improved [2]. However, these progresses accompanied by other issues such as decreasing the supply voltage levels and increasing clock frequency will affect the reliability of the microprocessors and lead them to more susceptible ones to transient faults. Transient faults are the most probable threats in modern microprocessors which have been induced by energetic particle strikes, such as high-energy neutrons from cosmic rays, and alpha particles from decaying radioactive impurities in packaging and interconnect materials [3].

The transient faults can manifest it self in three major types [4], [5]: 1) benign errors, 2) control-flow errors, and 3) data errors. It has been shown that, about 10% of them lead to benign errors [4]. For example, errors occurred in invalid blocks of the cache memories, or errors emerged in the blocks which will be replaced in near future. Also, considerable fraction of them, between 33% and 77%, reflects control-flow errors, such as possible errors in program counter (PC), address circuits, steering and control logic, or any kind of memories' controllers. Other transient faults lead to data errors, for instance, errors in the storage structures such as different level of cache memories, main memory, or registers [4], [5], [6].

The main concentration of this paper is on Control-Flow Errors (CFEs). According to their incidence, ability of dealing with them is obviously necessary, especially for safety critical applications. A CFE can affect the correct execution of the programs as sequence of the instructions [7]. CFEs can be divided into two types: inter nodes and intra nodes. An intra node CFE is an illegal movement within a node (basic block), and an inter node CFE is an illegal

movement between two nodes, or an illegal movement from a node to an unused spaces of memory called partition block.

Control-Flow Checking (CFC) is a key method for monitoring the flow of a program which partitions the program code into basic blocks (that are branch free groups of instructions terminated by a branch), and then adds redundant hardware/software components for checking the execution flow of the program. Several methods have been designed for detecting each type of the CFEs in a program code. Hardware-based methods use an external hardware like watchdog (checker) processor to monitor state and performance of the main (master) processor [8], [9], [10], [11]. On the other hand, software-based techniques make use of redundant software instead of hardware and are based on signature assignment to each basic block. Signatures are updated at runtime and next compared with the original ones which were calculated at compile time [7], [12], [13], [14].

Because the area- and power-efficiency of the new and modern processors, enable them to stay within their area, power and thermal budgets, adding redundant hardware for fault tolerance or error handling would undermine the obtained benefits of the modern processors [11], [15], [16]. Also, it should be noted that these types of hardware methods have been implemented through monitoring the accesses of the master processor to main memory. However, this method cannot be applied on modern processors because of having several levels of cache memories [8]. Therefore, making use of redundant software is a more convenient option than redundant hardware, especially for current architectures.

Methods for CFEs detection has been widely studied in the literature [17], [18], [19], [20]. However, unfortunately only a few published works focused on correcting the detected CFEs [6], [21]. Moreover, correcting the CFE is not sufficient, and there is no guarantee that the program will not fail since there may be some data errors generated by the CFEs. Therefore, the data errors arisen because of CFE occurrence should also be recovered before, after or during the phase of correcting the CFE. To correct data errors, programs need to gets some checkpoints during code execution, and then restore and re-execute from the last checkpoint. However, this may not be feasible in any application, because getting checkpoint, re-storing program states and re-executing the program are potential of imposing significant area cost and latency [6].

This paper proposes two techniques for automatic CFEs and data errors correction, as similar as some previous works [6], an error-handler function is organized at design time in order to correct CFEs. Moreover, added instructions in each basic block and also the error-handler function are designed and specified so that most of the data errors observed because of CFEs can be recovered during correction phase. Extracting control- and data-flow graphs from program code is the pre-requisite step in both of the proposed methods, and differences between them are appeared only due to structure of the error-handler function. The first one, Control-flow and Data Errors Correction using Data-flow Graph Consideration (CDCC), can correct CFEs and data errors by re-executing the program from the basic block after which all of the affected variables by the CFE will be re-initialized. The second technique, Miniaturized Check-Pointing (MCP), tries to correct both types of the errors by implementing a partial check-pointing method and providing some shadow variables for original ones that always store the correct values of them.

To evaluate the proposed techniques, five well-known benchmarks (with Solaris standard) are utilized. These benchmarks were run on a simple processor with a real operating system (Open Solaris9), simulated by a functional full-system simulation infrastructure, SIMICS [22]. The results of injecting about 8000 transient faults into several executable points of the programs reveal that 93.2% and 90.5% of the CFEs are detected and corrected by CDCC and MCP methods without any data errors generation, respectively. The experimental results show that performance and memory overheads of CDCC and MCP are considerably less than previous related work like ACCED [6] which is based on duplication methods for correcting the data errors.

The structure of this paper is as follows: section 2 gives related work and introduces terminologies. Section 3 describes the points which are disregarded by the previous methods as their weaknesses. The proposed techniques are explained in section 4. Simulation environment and experimental results are presented by section 5. Finally, section 6 concludes the paper.

2. Related Work

Various works related to detect and correct CFEs have been presented yet. This section only concentrates on software based methods due to the similarities in the basis of idea exist between proposed and previous methods.

The Block Signature Self Checking (BSSC) technique presented in [17] is purposed to check the flow of a program between branch free blocks (basic blocks) by defining a signature for each basic block and check it at the end of the blocks. In this technique, at the beginning of the blocks, a subroutine is called and address of the first instruction in the node is pushed into top of the stack as the signature, or it is stored in a static variable. A subroutine at the end of the basic block compares the embedded signature with the signature stored by the entry routine. Therefore, it can detect an illegal jump between two nodes, caused by a CFE. Also, in Error Capturing Instructions (ECI) technique [17], errors which cause erroneous execution in the unused spaces and data spaces are detected. This technique fills unused space (partition block) with branch instructions to an infinite loop or with software interrupt instructions.

The Control-flow Error Detection through Assertions (CEDA) technique is presented in [7]. In this technique like previous techniques, extra instructions are automatically embedded into the program at compile time in order to update run-time signatures continuously, and to compare them with pre-assigned value. Previous techniques insert multiple instructions in the basic blocks whenever they want to update the run-time signature. However, this technique inserts fewer instructions than the previous works, because of calculating signatures differently. Therefore, this technique has less overhead and is more efficient than the prior works. Decreasing overheads and increasing effectiveness are main purposes of this approach.

The Automatic Correction of Control-flow Errors (ACCE) technique [6] uses CEDA technique for CFEs detection. After the detection phase, a predefined function called error-handler is automatically executed, and the program control is transferred to the function and then to the basic block in which the illegal jump has been occurred. An

extension of ACCE, Automatic Correction of Control-flow Errors with Duplication (ACCED), has been implemented for data error detection and correction through duplicating instructions. Although this method can detect and correct some data errors [6], however, some sort of them cannot be detected, and also, the area and memory overheads of this method are really high that maybe intolerable for some applications.

3. Limitations of the Prior Techniques To Correct Data Errors

As mentioned in previous sections, detecting and correcting the CFEs are not sufficient and applicable, while some data errors, generated by the CFEs, remain active in the system. figure 1 illustrates how the data errors can be generated due to a CFE, and also shows the custom structure of the basic blocks in most of the software techniques. Regarding to figure 1 (b), adding instructions at the beginning and at the end of the basic blocks to check and update the signatures is the principal reason to reach the highest coverage for detecting the inter-node CFEs (*Branch 1*). In order to detect a number of intra-node CFEs, a node can be divided into sub-nodes, thus detecting intra-node CFEs which now become inter-node ones. Four shaded lines at the beginning and at the end of each block represent the added instructions in two of the recent published methods [6], [7] having the highest detection coverage and the least overhead. Conditional branch instructions are utilized for checking statements. Also, comparing the run-time and pre-defined values of the signature is used as its condition to detect any misbehavior. After checking phase, XOR operation updates the value of the run-time signature.

Regarding to figure 1 (a), after a CFE is detected, a function, called CFE-handler, can transfer the control to the block in which the CFE has occurred (*Branch 3*). This process is performed through finding the source basic block by comparing the value, stored in the signature register, with predefined values assigned to each block at design or compile time.

Unfortunately, this type of CFE correction, used by the previous published technique, is potential of arising data errors in the system. The data errors can happen because of executing the set of instructions resided in *Region 1* (which are not executed at regular time), or the set of instructions resided in *Region 2* (that are performed wrongly) or the set of instructions resided in *Region 3* (which are executed additionally). So, effectiveness of the correction process is reduced, because of data errors which are likely to observe in the output.

4. The Proposed Techniques

Due to the importance of data errors, there are some works in the literature focusing on correction of them based on instruction duplication, [6], [21]. Unfortunately, this method for data errors correction imposes high and intolerable overheads because of duplicating and comparing processes.

First the common properties of two proposed techniques are mentioned. To develop basis of the ideas, two flow graphs, used by the proposed techniques, should be introduced. The first one is Control-Flow Graph (CFG) which is a graph with set of nodes (basic blocks) and directed edges that show the right transmission among basic blocks. The second one is Data-Flow Graph (DFG), a directed edges graph with set of nodes (operations and variables) and edges which present data dependencies among variables. These two graphs should be extracted from the relations among basic blocks and variables of the program to implement details of the added instructions and the error handler functions.

The CFE detection methods used by the MCP and the CDCC are quite similar to CEDA, ACCE [6], [7]. Figure 2 shows the added instructions to each basic block. The differences between the proposed methods, appeared in figure 2, are only because of applying different types of correction. In correction phases of the methods, signatures of the source and the destination basic blocks are required. Therefore, these signatures are maintained within two registers (Source Signature Register and Destination Signature Register) during the execution. Source Signature

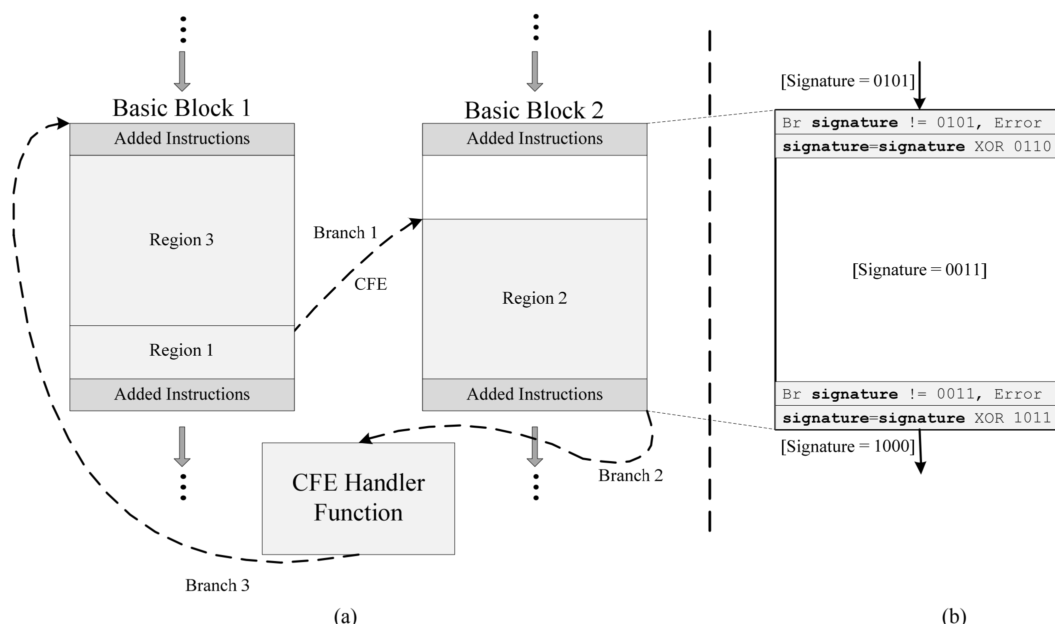


Figure 1. (a): Data Errors Generation Due to a CFE, (b): Custom Structure of a Basic Block in Most of the Software Techniques

Register (SSR) is a run-time register which is continuously updated, and finally stores the signature of the basic block in which a CFE has occurred. Also, Destination Signature Register (DSR) is a run-time register which is continuously updated, and finally stores the signature of the basic block to which the control is incorrectly transferred. Moreover, in correction phase of the MCP technique, the variables known as shadow variables have been used for storing copies of the original variables.

The signature of the destination basic block (that control is transferred to it illegally) is stored in DSR at the end of the nodes. When an illegal jump occurs to middle of the node, the stored value in DSR cannot be reliable, if the destination signature is stored at the beginning, because the signature of the previous basic block (source) was stored in DSR (this value can mislead the CFE-handler function) while the added instructions at the end of the basic block change this situation and store the true value in the DSR.

4.1. The Proposed CDCC Technique

When a CFE is detected through added instructions, the control is transferred to CFE-handler function. This function is implemented by considering the DFG and CFG of the program at design time. The signatures of the source and destination basic blocks are given to CFE-handler function as inputs. This function can relocate the control to the nearest block from which re-executing the program corrects the CFE, and all of the affected variables between source and destination will be re-initialized.

Figure 3 (a) shows three basic blocks from the set of basic blocks in a program code as well as the DFG extracted from data dependencies among variables in these basic blocks. figure 3 (b) illustrates the process of the correction used by the proposed techniques.

Regarding to them, if *CFE1* has occurred in *basic block 2* and the control is transferred from *basic block 2* to *basic*

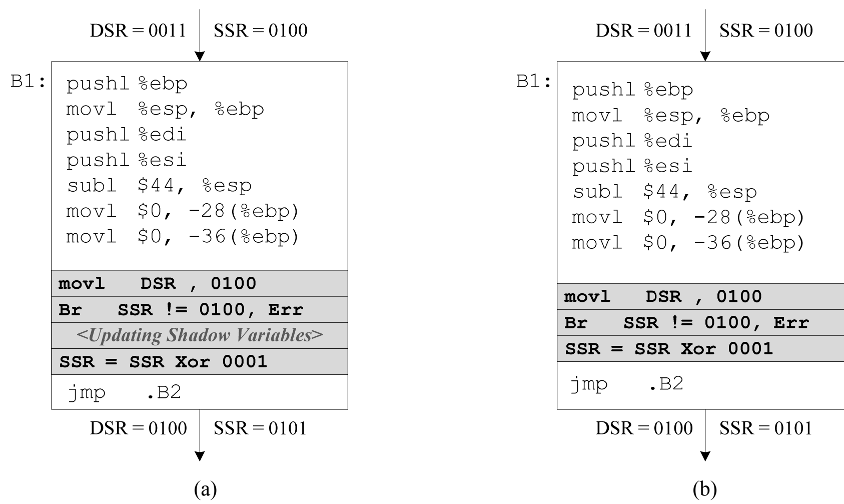
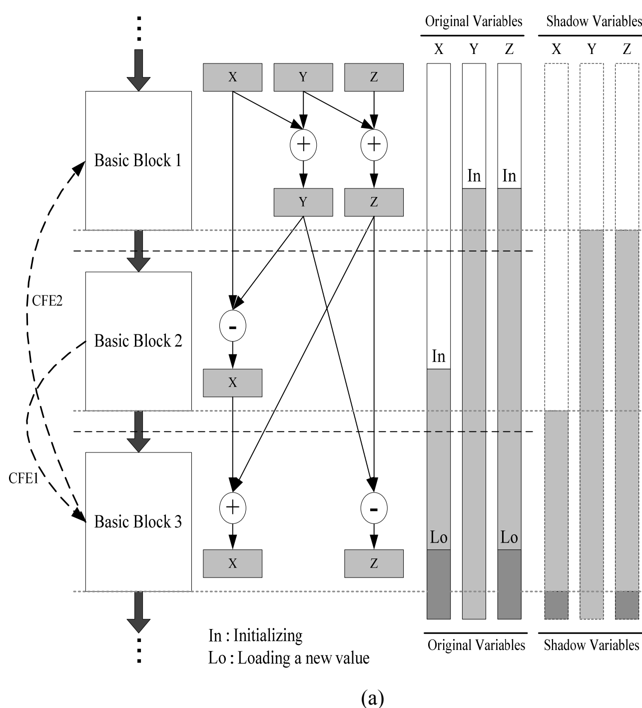


Figure 2. Illustration of Added Instructions in The Proposed Techniques: (a): MCP (b): CDCC



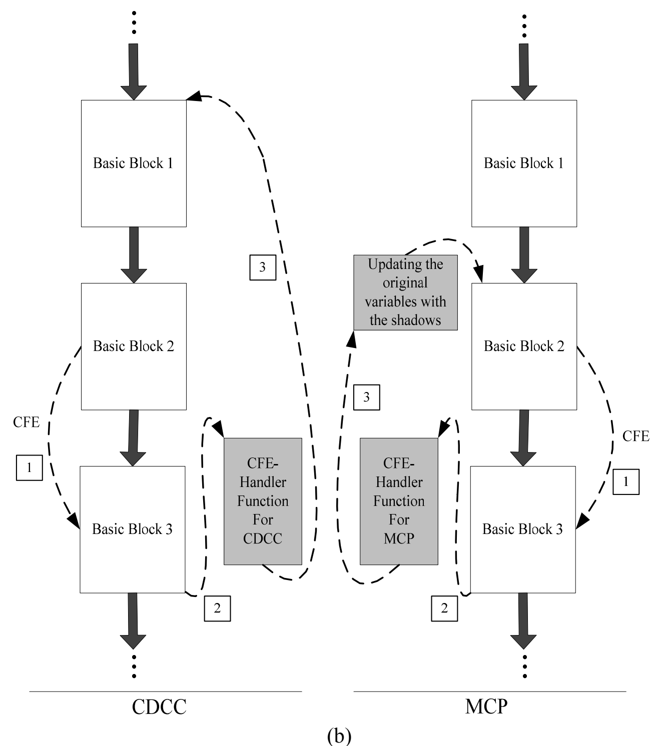


Figure 3. (a): CFG and DFG Generated from Program Code, (b): Differences between MCP and CDCC Methods

block 3 (step 1 in figure 3 (b)), then the values stored in variables *X* and *Z* cannot be reliable, because of the problems previously explained. For example, suppose that the source basic block is *basic block 2* and the destination one is *basic block 3,* also the variables modified by the CFE (*X* and *Z*) are initialized in *basic block 1* and *basic block 2*. For CFE and data errors correction, the control should be transferred to *basic block 1* (step 3 in figure 3 (b)). Therefore, the modified variables are re-initialized and their corresponding computations are re-executed after this transmission. By re-executing the code from *basic block 1*, the first value which was stored in variable *Z* is re-loaded again. Also, after completing *basic block 1* and in *basic block 2*, the first value of *X* is re-loaded.

Another example is when CFE2 occurs, then the source basic block is *basic block 3* and the destination one is *basic block 1*. The variables affected by this CFE are *X*, *Y*, and *Z*. The initialization of *X* is done in *basic block 2*, and the initialization of variables *Y* and *Z* are done in *basic block 1*. Hence, returning to *basic block 1* leads to load the initialization values to variables and re-execute computations by which the variables had been used.

In this technique for detection and correction of illegal jumps to unused space (partition block), the partition block is filled-up with branch instructions to CFE-handler function. Zero (Null) is reserved as the destination signature value for the partition block to distinguish it from the other blocks in the program code. If the illegal jump occurs to it, The CFE-handler function ignores the destination, because it contains no computation related to the program.

4.2. The Proposed MCP Technique

In the first section, some problems (potential of imposing high overhead and high latency) of checkpoint-based methods were explained. In this type of error correction, the program gets checkpoints at some points during code execution. At these times, the values stored in variables (such as registers and memory blocks) should be sustained in shadow variables. Then, for CFEs and data errors correction, it needs to re-execute from the last trustable checkpoint, after loading correct values stored in shadow variables to original ones.

Through the MCP, the shadow variables always contain the true values of the original ones. Moreover, only some of the shadow variables are updated at the end of each basic block in which the corresponding original variables has been modified. This optimization directly leads to noticeable reduction in the overheads of checkpoint-based methods.

The segments of updating the shadows can be specified with respect to the information provided by the DFG of the program. Regarding to figure 3, the time of updating the shadow variables with the original ones is clear. Suppose that variables *Y* and *Z* are initialized in *basic block 1*, and variable *X* is initialized in *basic block 2*. For example, the value of variable *Y* is only changed in *basic block 1.* Therefore, the shadow of *Y* should be updated only at the end of this basic block (instead of updating at the end of the all basic blocks).

As described in the previous sections, the CFE can change the stored values in variables used in computations in the source and the destination, and it can lead to data errors generation in the output. After detection phase, the control is transferred to CFE-handler function (step 2 in figure 3 (b)). At this time, the signatures of the source and destination basic blocks are already available in SSR and DSR, respectively. These two values are given to CFE-handler function as inputs. This function can update the affected original variables in the source and the destination with shadow ones, and also it can determine address of the first instruction in the source basic block. Finally, the control is transferred to this address (step 3 in figure 3 (b)) and the code is re-executed from this point. Consequently, both of the CFE and the generated data errors can be corrected.

For improving the MCP, the temporal and local variables can be ignored, and the process of making shadows for them can be omitted. By this improvement, shadows are only made for global variables which are alternatively used in the program code.

4.3. CFE-Handler Functions

Figure 4 (a) and (b) show implementation samples of the CFE-handler function defined for the CDCC and MCP, respectively.

The CDCC function first determines the source basic block by comparing the value stored in SSR with the signatures assigned to each basic block at design time. Then, the control is transferred to sub-sections which are separately defined for each source basic block. In these subsections the destination basic block can be determined as similar as determining the source one, and finally the control can be transferred to the basic block in which the first initialization of the affected register is done. This transition can be performed by conditional branches to the first instruction of the basic blocks.

When an illegal jump occurs to the CFE-handler function statements, the function can gives the control back to the source basic block, by executing the first subsection. The last lines of the subsections (jump instructions to the first line of the function) were defined to correct this type of CFEs.

Unlike the CDCC, the function written for the MCP determines the destination basic block at the first. Then the affected original variables in the destination are updated with shadow ones. After that, the source basic block is determined and the affected original variables in the source are updated with the shadow ones too; and finally, the control is transferred to the source basic block. Consequently, the affected original variables store true values and the program re-executes from the source basic block in which the CFE has occurred.

In the case of fault occurrence in shadow variables, the erroneous values of shadow variables are overwritten at the end of each basic block. Following information explain the details of this issue: Through the MCP, the shadow variables contain true values of the original ones in all the cases except the case where intra-block CFEs are occurred during the time between loading new values into original variables and execution of the last instruction in corresponding basic block. Similar to the previous techniques, the proposed ones cannot detect the intra-block CFEs. Therefore, the proposed techniques try to detect inter-block CFEs and correct the data errors raised because of their effects.

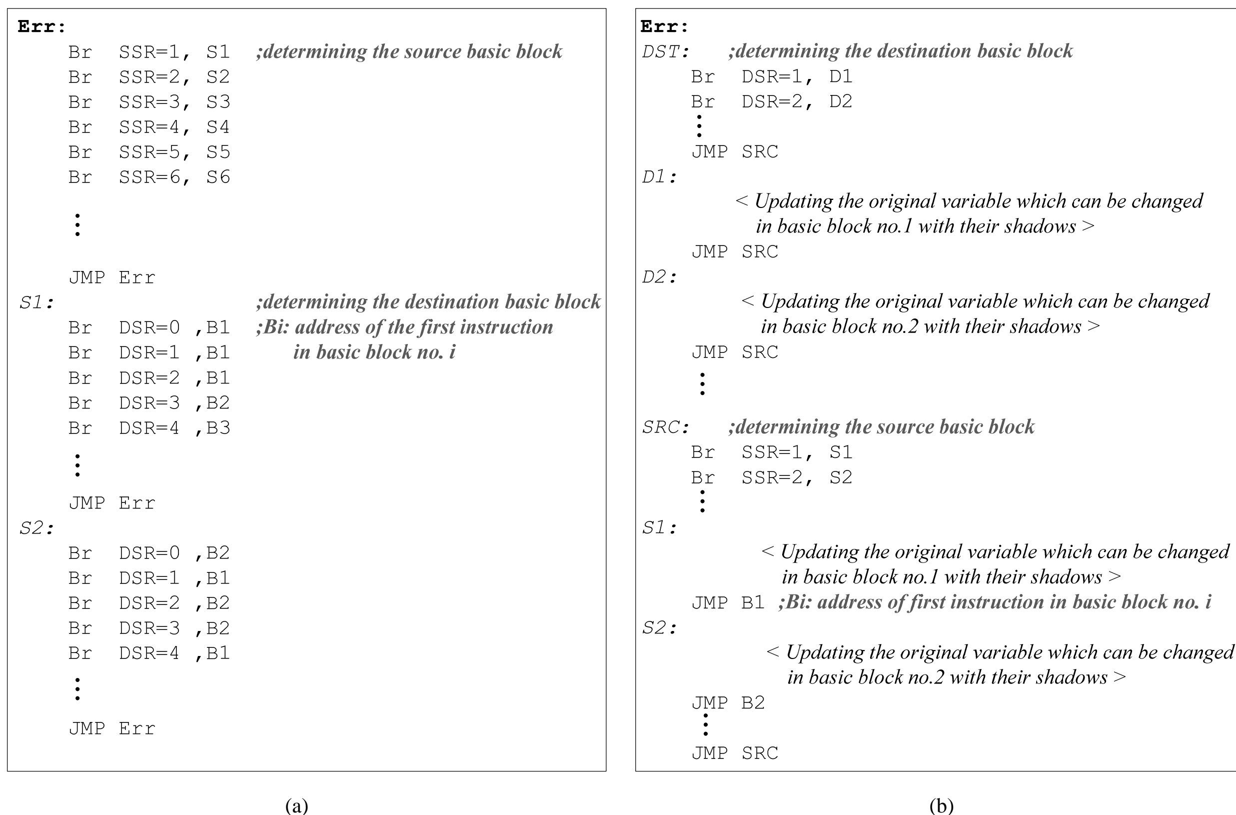


Figure 4. Implementation of the CFE-Handler Function for: (a): CDCC, (b): MCP

In the case of inter-block CFEs (as illustrated by figure 3 in the manuscript), the shadow variables are updated at the end of each basic blocks. While the average length of each basic block is about 9 instructions, the shadow variables are updated just after about 4 instructions (on average) after the modifications of original ones. When the effects of intra-block CFEs have not been considered, the control flow of execution in a basic block is not affected. Therefore, the values of shadow variables cannot be changed during the execution.

If an illegal jump happens to the CFE-handler function, the control should be given back to the source basic block in which the original variables were affected, after updating with the shadows. This is performed by executing the subsection organized for determining the source basic block. Therefore, the jump instructions to the *SRC* were inserted at the end of the subsections.

While the proposed techniques and previous works do not change the sequence of instructions during execution, memory accesses (in terms of addresses and sequences) will be quite similar. Therefore, the number and sequence of cache hits and misses are similar for our design decisions and previous works. In order to validate this claim, addresses of consecutive accesses to memory have been observed for previous works and proposed techniques. The observation has been shown that the sequences of memory accesses are completely same. Correspondingly, the effects of memory hierarchy on the results of previous and proposed works will be same.

While SSR/DSR contains the number of basic blocks from/to which an illegal jump occurred, changing the values of these registers (by means of faults) leads to wrong recognition of source/destination basic block. However, this misdiagnosis cannot affect the normal output of the program. Regarding the structure of CFE handler function (figure 4 of the manuscript), faults in these two registers misleads CFE handler function to select the basic block to which the final jump should be occurred. This issue may cause that the program re-execute from a basic block which is farther than the main basic block to the end of the program. Therefore, faults in SSR or DSR lead to increased correction latency and increased normal execution time of the program.

4.4. Optimization of the CFE-Handler Function

The CFE-Handler functions of the proposed methods are organized so that the detected CFEs are corrected with minimum correction latency. However, in some applications, imposing less memory overhead may be more important than the other issues, due to the area-efficiency of their designs.

Therefore, the structures of these functions should be optimized under the memory constraint. As shown by figure 5, the targets of the final branches, used in the last phase of the correction, are specified taking to account the pair of the source and destination basic blocks. So, to reduce the number of instructions in the functions, the phase of determining the destination basic block can be omitted.

For example, regarding to figure 5, if the source basic block of an illegal jump is *basic block 1*, then, the target of the final branches is also *basic block 1*, independent of the destination basic block. In the other case (for example the forth line of the matrix in figure 5), considering the topmost basic block, nearest basic block to the beginning of the program, from the set of the targets for one specific source basic block causes that one of the phases in the correction

process (checking the value of the DSR for determining the destination) is omitted.

These optimizations in designing the structures of the functions lead to effective improvements in the percentages of the memory overhead of the proposed techniques, presented by the next section.

5. Experimental Results

In order to evaluate the proposed techniques, a functional simulation infra-structure called SIMICS [22] has been used as the simulation environment, through providing a simple processor simulated in SPARC V9 ISA with a real operating system (Open Solaris 9). Also, the behaviors of five well-known benchmarks have been studied. Table 1 summarizes the attributes and description of each benchmark.

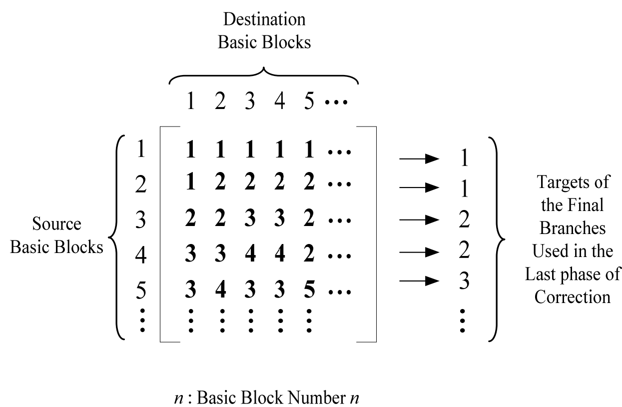


Figure 5. Schematic of an Algorithm for Reducing the Memory Overhead of the CFE-Handler Function

Simulation-based error injection method has been used and about 8000 errors have been injected on several executable points of the programs through modifying the assembly code of the source files. Three error types are considered [6], [7], [12], [13], [14]:

- Branch deletion: it had occurred when one of the branch instructions consisting unconditional ones such as *jmp*, *cal*, and *ret*, and conditional ones such as *jne*, *jl*, *jle*, *jg*, and *jge*was randomly replaced by a NOP instruction.
- Branch insertion: it had occurred when an unconditional branch (*jmp*) was randomly inserted into the programs.
- Branch operand changes: it had occurred when the immediate field of the branch instructions was randomly corrupted.

During fault injection experiments, all of the points in the programs that can be source of an illegal jump are considered. In other words, three different types of faults (branch insertion, deletion and operand changes) are exhaustively injected into program codes. Therefore, fault injection is applied on all of the points in the program code that have the potential of CFE occurrence. In these experiments, target (place) and time of injected faults are considered as uniform distributions since fault causes are almost from environment disturbances. The granularity of fault injection time is set to instruction cycle. Like previous related works, one CFE error for each experiment is assumed in the paper.

Table 2 shows the correct ability of the programs with the proposed techniques in compare to correct ability of the programs with ACCED and Complete Check-Pointing (CCP). In order to compare the proposed techniques with the methods implemented based on complete check-pointing techniques; a method (CCP) has been developed and applied on the benchmarks, with recognition of the check-point based methods' properties. Correct ability shows the power of the technique to correct the CFEs and data errors. It is said that a CFE (and generated data errors) has been corrected if the program exits normally with correct results. About half of the errors had led to segmentation faults and were detected by the Operating System (OS). So, these detected CFEs were not considered as the errors that can be detected and corrected by the techniques. As shown by table 1, about 93.2%, and 90.5% of the injected errors return correct output with CDCC and MCP, respectively. The correct ability of MCP is a little less than the CDCC, because of high susceptibility of the program code segments in which the shadow variables are updated. Moreover, the Correct ability of the CCP is the least among all related techniques, due to using larger size of check-pointing (updating) segments.

Furthermore, to compare the effectiveness of the techniques, three parameters have been characterized. The first factor called *Memory overhead (M.O.)* showing the amount of memory imposed to the program by the methods (because of the number and length of the added instructions), is computed by:

$$\mathrm{M.O.} = {\Delta\text{Initial Memory Usage}}/{\text{Initial Memory Usage}} \quad (1)$$

The second parameter is *Performance overhead (P.O.)* that reveals the amount of performance degradation due to methods' operation, and is estimated by:

$$\mathrm{P.O.} = {\Delta\text{ Performance}}/{\text{Initial Performance}} \quad (2)$$

To compare previous works with design decisions in terms of performance, number of clock cycles, needed to complete each program in the benchmark set, is calculated. The results are normalized to the results of base execution

Table 1. Benchmarks Description

Benchmarks	**Quick Sort (QS)**	**Matrix Multiplication (MM)**	**Bubble Sort (BS)**	**Linked List Insertion (LL)**	**Fast Fourier Transform (FF)**
Description	Sorts a 100-elements array of integers.	Multiplies two 4×4 matrixes.	Sorts a 100-elements array of integers.	Adds10elements to a linked list.	Computes the Discrete Fourier Transform.
# of Instructions	257	578	142	276	406
# of Basic Blocks	31	55	16	32	38
Basic Block Sizes	8.3	10.5	8.9	8.6	10.7

without any added instruction.

Although the MCP needs few instructions to be executed after CFEs, however, the shadow variables should be updated at the end of each basic block. This operation is applied only on the shadow variables that their corresponding original variables have been changed. Moreover, this operation cannot be parallelized with the normal execution, because it is possible for the original values to be changed during updating operation. Therefore, serial updating the shadow variables causes that the performance overhead of MCP is always higher than CDCC.

Table 3 demonstrates the values of the two described parameters obtained after applying each related methods on the programs. In order to show the effects of the area (memory) optimization, explained in section 4.4, the percentage of the memory and performance overhead of the enhanced proposed techniques ($CDCC^+$, and MCP^+) are also compared with the other related techniques. As shown in table 3, the memory and performance overheads of the proposed technique are lower than the ACCED and CCP. The memory (performance) overhead of the ACCED and CCP are comparatively (more than 100%) higher than the proposed techniques because of adding (executing) duplicated instructions and adding (executing) the set of instructions used for comparing the results to obtain correct output.

Finally, to give a general comparison among all the methods (as similar as some previous works [6], [7]), which also takes into account all of the impressive parameters, a metric called *Correction Coverage per Cost (CCC)* is defined to estimate the efficiency of the methods:

$$\text{CCC} = {}^{\text{Correction Coverage}}/_{\text{Overall Cost}}, \quad (3)$$

where overall cost is calculated depending on the type of the applications on which the techniques are applied. Two impact factors have been defined for costs: α and β. The impact factors of the *performance cost* and *Memory cost* in the system is estimated by α and β, respectively. In each system, sum of these impact factors should be equal to one ($\alpha + \beta = 1$). In some applications the performance cost is more important than the memory one. In these situations, the value of α should be rather than β. On the contrary, if the importance of the memory cost is higher than the performance one, the value of β should be rather than α. The overall cost is estimated as follow:

$$\text{OverallCost} = (\alpha \times \text{P.O.}) + (\beta \times \text{M.O.}) \quad (4)$$

Without loss of generality, let $\alpha = \beta = 0.5$, and this means that the importance of the performance and memory cost are equal in the estimation.

ACCED is the extended version of ACCE technique. While the ACCE is not capable to detect and correct data errors, the ACCED can detect and correct about 96% of data errors via instruction duplication technique. However, the memory and performance overheads of the ACCED are about 318%, and 215% respectively. These high overheads are imposed because of adding/executing duplicated instructions and adding/executing the set of instructions used for comparing the results. The proposed techniques are intended to enhance the method efficiency of the data error correction techniques. These techniques (CDCC and MCP) and their extensions ($CDCC^+$ and MCP^+) reduce the mentioned overheads significantly (about 100% reductions) with negligible negative impacts on correction coverage (about 3%). figure 6 compares the level of the efficiency among the related methods.

According to that, the *CCC* of the proposed techniques is

Table 2. Experimental Error Injection Results

Benchmarks	Original		ACCED		CCP		CDCC		MCP	
	Wrong Results (%)	*Correct Results (%)*	*Wrong Results (%)*	*Correct Results (%)*	*Wrong Results (%)*	*Correct Results (%)*	*Wrong Results (%)*	*Correct Results (%)*	*Wrong Results (%)*	*Correct Results (%)*
QS	91.4	8.6	2.6	97.4	9.8	90.2	5.1	94.9	7.8	92.2
MM	93.7	6.3	4.6	95.4	17.8	82.2	8.1	91.9	10.4	89.6
BS	93.4	6.6	3.8	96.2	14.1	85.9	6.4	93.6	9.4	90.6
LL	93.3	6.7	3.1	96.9	11.9	88.1	5.8	94.2	8.6	91.4
FF	94.2	5.8	5.2	94.8	20.0	80.0	8.8	91.2	11.2	88.8
Average	**93.2**	**6.8**	**3.9**	**96.1**	**14.7**	**85.3**	**6.8**	**93.2**	**9.5**	**90.5**

Table 3. Comparison of the Memory and Performance Overheads

Benchmarks	ACCED		CCP		CDCC		MCP		$CDCC^+$		MCP^+	
	*M. O. (%)**	*P. O. (%)***	*M. O. (%)**	*P. O. (%)***	*M. O. (%)**	*P. O. (%)***	*M. O. (%)**	*P. O. (%)***	*M. O. (%)**	*P. O. (%)***	*M. O. (%)**	*P. O. (%)***
QS	365.2	245.7	385.1	257.0	295.2	84.5	316.8	114.1	106.9	84.5	147.5	114.1
MM	284.0	190.2	339.4	226.3	283.3	75.2	303.6	99.3	96.6	75.2	130.9	99.3
BS	325.6	223.0	352.7	234.6	287.0	78.9	309.9	102.7	99.2	78.9	135.2	102.7
LL	341.3	232.5	366.5	248.9	291.5	82.0	313.0	108.5	103.0	82.0	141.1	108.5
FF	275.5	183.3	329.0	227.3	280.1	73.5	298.1	96.0	93.1	73.5	127.9	96.0
Average	**318.3**	**214.9**	**354.5**	**238.8**	**287.4**	**78.8**	**308.3**	**114.1**	**99.8**	**78.8**	**136.5**	**114.1**

* Memory Overhead Percentage
** Performance Overhead Percentage

considerably more than the *CCC* of the ACCED and CCP. It brings this concept that the proposed techniques are more efficient in compare to the previous techniques, especially for applying on current and modern architectures, where imposing less overhead is another major concern along with insusceptibility against any types of faults and errors. These enhancements have been achieved due to the noticeable reduction in the imposed overheads, and due to preparing high correction coverage for two types of the errors (CFEs and data errors).

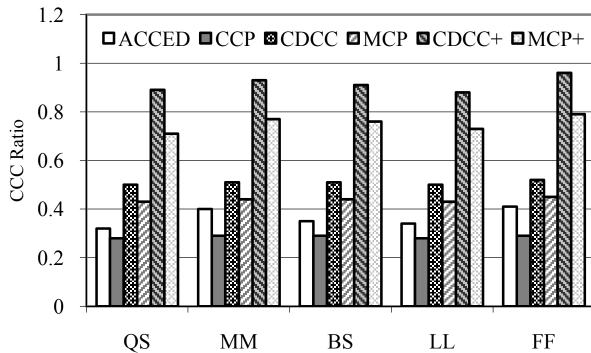


Figure 6. Comparison of the Method Efficiency

6. Conclusions

In this paper, two efficient software techniques to detect and correct control-flow errors (CFEs) were proposed. These techniques were implemented through considering both of control-flow and data-flow graphs, extracted from the program code, at design time. Neglecting the effects of the data errors arisen because of the CFE would undermine the effectiveness of the previous techniques presented in order to detect and correct only the CFEs. In order to develop the proposed techniques, first a signature was assigned to each node for CFE detection, and then some instructions were specified for calculating and checking them at runtime. A function, defined for automatic correction of the CFEs and data errors, was implemented through considering the information about the relations among basic blocks and variables of the program, provided by control-flow and the data-flow graphs of the program code. Error injection experiments showed that the proposed techniques, when applied on the programs, produce correct results in over 93% and 90% of the cases, respectively. The latency and the additional memory required for correcting the CFEs and the data errors are considerably less than the duplication based and check-pointing based methods which have been recently published. A metric for estimating efficiency of the techniques was defined, and it was shown that the proposed techniques are more efficient compared to duplication-based and checkpoint-based methods.

References

[1] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "SWIFT: Software Implemented Fault Tolerance," *Proc. IEEE Intl Symp. Code Generation and Optimization,* pp. 243-254, 2005.

[2] J. Srinivasan, V. Adve, P. Bose, and J. Rivers, "The Impact of Technology Scaling on Lifetime Reliability," *Proc. IEEE Intl Conf. Dependable Systems and Networks*, pp. 177-186, 2004.

[3] J. Ray, C. Hoe, and B. Falsafi, "Dual Use of Superscalar Data Path for Transient-Fault Detection and Recovery," *Proc. ACM/IEEE Intl Symp. Micro-architecture,* pp. 214-224, 2001.

[4] J. Ohlsson, M. Rimen, and U. Gunneflo, "A Study of the Effects of Transient Fault Injecting into 32-bit RISC with Built-in Watchdog," *Proc. IEEE Intl Symp. Fault Tolerant Computing,* pp. 316-325, 1992.

[5] U. Gunnejlo, J. Karlsson, and J. Torin, "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-Ion Radiation," *Proc. IEEE Intl Symp. Fault Tolerant Computing,* pp. 340-347, 1989.

[6] R. Vemu, S. Gurumurthy, and J. Abraham, "ACCE: Automatic Correction of Control-flow Errors," *Proc. IEEE Intl Conf. Test,* pp. 1-10, 2007.

[7] R. Vemu, and J. Abraham, "CEDA: Control-flow Error Detection through Assertions," *Proc. IEEE Intl Symp. On-Line Testing,* pp. 151-158, 2006.

[8] A. Rajabzadeh, and S. Miremadi, "CFCET: A Hardware-Based Control Flow Checking Technique in COTS Processors Using Execution Tracing," *Elsevier Journal of Microelectronics Reliability,* vol. 46, no. 3, pp. 959-972, 2006.

[9] N. Farazmand, M. Fazeli, and S. Miremadi, "FEDC: Control Flow Error Detection and Correction for Embedded Systems without Program Interruption," *Proc. IEEE Intl Conf. Availability, Reliability and Security,* pp. 33-38, 2008.

[10] I. Majzik, A. Pataricza, W. Hohl, J. Honig, and V. Sieh, "A High-speed Watchdog Processor for Multitasking Systems," *Proc. IEEE Intl Symp. Microcomputer and Microprocessor Applications,* pp. 80-85, 1994.

[11] A. Meixner, and D. Sorin, "Detouring: Translating Software to Circumvent Hard Faults in Simple Cores," *Proc. IEEE Intl Conf. Dependable Systems and Networks*, pp. 80-89, 2008.

[12] N. Oh, P. Shirvani, and E. McCluskey, "Control-flow Checking by Software Signatures," *IEEE Trans. Reliability,* vol. 51, no. 1, pp. 111-122, 2002.

[13] Y. Sedaghat, S. Miremadi, and M. Fazeli, "A Software-based Error Detection Technique Using Encoded Signature," *Proc. IEEE Intl Symp. Defect and Fault Tolerance in VLSI Systems,* pp. 389-400, 2006.

[14] P. Bernardi, L. Bolzani, M. Rebaudengo, M. SonzaReorda, F. Vargas, and M. Violante, "On-line Detection of Control-Flow Errors in SoCs by Means of an Infrastructure IP core," *Proc. IEEE Intl Conf. Dependable Systems and Networks,* pp. 50-58, 2005.

[15] A. Shye, T. Moseley, V. Reddi, J. Blomstedt, and D. Connors, "Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance," *Proc. IEEE Intl Conf. Dependable Systems and Networks,* pp. 40-46, 2007.

[16] A. Shye, J. Blomstedt, T. Moseley, V. Reddi, and D. Connors, "PLR: A Software Approach to Transient Fault Tolerance for Multi-Core Architectures," *IEEE Trans. Dependable and Secure Computing,* vol. 8, no. 2, pp. 1-12, 2008.

[17] S. Miremadi, J. Karlsson, U. Gunneflo, and J. Torin, "Two Software Techniques for On-Line Error Detection," *Proc. IEEE Intl Symp. Fault-Tolerant Computing,* pp. 328-335, 1992.

[18] O. Goloubeva, M. Rebaudengo, M. Reorda, and M. Violante, "Soft-error Detection Using Control Flow Assertions," *Proc. IEEE Intl Symp. Defect and Fault Tolerance in VLSI Systems*, pp. 581-588, 2003.

[19] H. Ghasemzadeh, S. Miremadi, and A. Ejlali, "Signature Self Checking (SSC): A Low-Cost Reliable Control Logic for Pipelined Microprocessors," *Proc. IEEE Intl Symp. Dependable Computing,* pp. 114-118, 2009.

[20] L. Aiguo, and H. Bingrong, "On-line Control Flow Error Detection Using Relationship Signatures among Basic Blocks," *Elsevier Journal on Computers and Electrical Engineering,* vol. 36, no. 3, pp. 132-141, 2010.

[21] T. Austin, "Diva: a Reliable Substrate for Deep Submicron Micro architecture Design," *Proc. ACM/IEEE Intl Symp. Micro-architecture,* pp. 196-207, 1999.

[22] Functional Full-System Simulation Infra-Structure (SIMICS), http://www.virtutech.com, May 2003.

Mohammad Maghsoudloo is Ph. D. candidate in department of Computer Engineering and Information Technology at Amirkabir University of Technology (AUT) (Tehran Polytechnic). He received his M.S. from Amirkabir University of Technology (AUT) (Tehran Polytechnic), and his B.S. from Mazandaran University, Iran, in 2011, 2009, respectively. His research interests include fault-tolerant computing, dependable computer architecture, chip multi-processors issues, application-level fault tolerance, cache memories and coherence problems, and dependability evaluation. He is a student member of the Computer Society of Iran (CSI), and IEEE Computer Society.
E-mail: m.maghsoudloo@aut.ac.ir

Hamid R. Zarandi received his B. Sc., M. Sc., and Ph. D. degrees all in department of computer engineering at Sharif University of Technology (SUT), Tehran, Iran, in 2000, 2002, and 2007, respectively. He is currently an assistant professor in computer engineering and information technology department at Amirkabir University of Technology (AUT) (Tehran Polytechnic), since 2006. His research interests include dependability evaluation using fault injection techniques, fault-tolerant computing, dependable computer architecture, high performance computing, and fault-tolerant embedded systems, on which he has published more than 90 referred conference and journal papers. Dr. Zarandi established the “Design and Analysis of Dependable Systems (DADS)” laboratory at Amirkabir University in 2006, and has chaired the laboratory since then. He is a member of the IEEE Computer Society, and the Computer Society of Iran (CSI).
E-mail: h_zarandi@aut.ac.ir

Navid Khoshavi is a Ph. D. student in department of electronic engineering and computer science at University of Central Florida. He received his M.S. from Amirkabir University of Technology (AUT) (Tehran Polytechnic), and his B.S. from Sepahan College of Education, Iran, in 2011, 2009, respectively. His research interests include fault-tolerant computing, dependable computer architecture, chip multi-processors issues, application-level fault tolerance, cache memories and coherence problems, and dependability evaluation. He is a student member of the Computer Society of Iran (CSI).
E-mail: navid.khoshavi@aut.ac.ir

Paper Handling Data:

Submitted: 09.09.2013
Received in revised form: 28.09.2014
Accepted: 11.10.2014
Corresponding author: Dr. Hamid R. Zarandi,
Department of Computer Engineering and Information Technology, Amirkabir University of Technology, Tehran, Iran.